UMIACS-TR-90-13 January 1990
CS-TR -2395

On Parallel Hashing and Integer Sorting

Yossi Matias*
Tel Aviv University

Uzi Vishkin†
Institute for Advanced Computer Studies and
Department of Electrical Engineering
University of Maryland
College Park, MD 20742
and
Tel Aviv University

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS-TR-90-13 January 1990
CS-TR -2395

On Parallel Hashing and Integer Sorting

Yossi Matias[*]
Tel Aviv University

Uzi Vishkin[†]
Institute for Advanced Computer Studies and
Department of Electrical Engineering
University of Maryland
College Park, MD 20742
and
Tel Aviv University

(Syn polylog(n)) (Concurrent Read Concurrent Write Parallel Random [illegible])

Abstract

The problem of sorting n integers from a restricted range $[1..m]$, where m is superpolynomial in n, is considered. An $o(n \log n)$ randomized algorithm is given. Our algorithm takes $O(n \log\log m)$ expected time and $O(n)$ space. (Thus, for $m = n^{polylog(n)}$ we have an $O(n \log\log n)$ algorithm.) The algorithm is parallelizable. The resulting parallel algorithm achieves optimal speed up. Some features of the algorithm make us believe that it is relevant for practical applications.

A result of independent interest is a parallel hashing technique. The expected construction time is logarithmic using an optimal number of processors, and Searching for a value takes $O(1)$ time in the worst case. This technique enables drastic reduction of space requirements for the price of using randomness. Applicability of the technique is demonstrated for the parallel sorting algorithm, and for some parallel string matching algorithms.

The parallel sorting algorithm is designed for a strong and non standard model of parallel computation. Efficient simulations of the strong model on a CRCW PRAM are introduced. One of the simulations even achieves optimal speed up. This is probably a first optimal speed up simulation of a certain kind. (KR)

*Part of this work was conducted while this author was visiting the University of Maryland.
†Partially supported by NSF grant CCR-890649 and ONR grant N00014-85-0046.

On Parallel Hashing and Integer Sorting

Yossi Matias
Tel-Aviv University

Uzi Vishkin[*]
Tel-Aviv University &
University of Maryland

Abstract

The problem of sorting n integers from a restricted range $[1..m]$, where m is super-polynomial in n, is considered. An $o(n \log n)$ randomized algorithm is given. Our algorithm takes $O(n \log\log m)$ expected time and $O(n)$ space. (Thus, for $m = n^{polylog(n)}$ we have an $O(n \log\log n)$ algorithm.) The algorithm is parallelizable. The resulting parallel algorithm achieves optimal speed up. Some features of the algorithm make us believe that it is relevant for practical applications.

A result of independent interest is a parallel hashing technique. The expected construction time is logarithmic using an optimal number of processors, and Searching for a value takes $O(1)$ time in the worst case. This technique enables drastic reduction of space requirements for the price of using randomness. Applicability of the technique is demonstrated for the parallel sorting algorithm, and for some parallel string matching algorithms.

The parallel sorting algorithm is designed for a strong and non standard model of parallel computation. Efficient simulations of the strong model on a CRCW PRAM are introduced. One of the simulations even achieves optimal speed up. This is probably a first optimal speed up simulation of a certain kind.

1 Introduction

Consider the problem of sorting n integers drawn from a given range $[1..m]$. A new randomized algorithm for the problem is presented. Its expected running time is $O(n \log\log m)$ using $O(n)$ space. The algorithm is parallelizable. The resulting parallel algorithm achieves optimal speed up. The result implies $o(n \log n)$ expected time and linear space for $m \leq 2^{n^{o(1)}}$.

[*]Partially supported by NSF grant CCR-8906949 and ONR grant N00014-85-0046.

More specifically, for $m = n^{\log^k n}$ (for any constant $k \geq 1$) we have $O(n \log\log n)$ expected time and $O(n)$ space. No such result is known for deterministic sorting, suggesting the following *fundamental open problem* : Is this an instance where randomization defeats determinism for sorting? The algorithm seems to be practical.

The paper employs two *Algorithmic techniques*:

1. A first randomized parallel hashing technique, which achieves optimal speed up and takes expected logarithmic time, is presented. The parallel hashing technique enables *drastic* reduction of space requirements in quite a few parallel algorithms for the price of using randomness. The technique is used in the parallel sorting algorithm. Such (serial) technique is also demonstrated in the serial sorting algorithm. The new parallel hashing technique, that results in trading space for randomness, is likely to have additional applications. We actually demonstrate it with a few examples.

2. The parallel sorting algorithm is designed for a strong and non standard model of parallel computation. New simulations of the strong model on a CRCW PRAM are introduced. Using one of the simulations the parallel sorting algorithm runs in optimal speed up also on a CRCW PRAM. Designing algorithms for strong and non standard models of computation and then translate them into standard models is a traditional methodology in computer science. We expect our parallel simulations to be helpful in this respect. The simulations are efficient: one of them even preserves optimal speed up.

1.1 Extant work

Sorting is a fundamental problem that has received much attention. [Knu73] gives several algorithms for sorting n objects drawn from an arbitrary totally ordered domain in $O(n \log n)$ time. There are also optimal parallel sorting algorithms in logarithmic time [AKS83] [Col86]. For the decision-tree model the $O(n \log n)$ time serial upper bound is best possible [AHU74].

Because of the central role that the sorting problem plays in computer science, numerous papers are devoted to study opportunities for improving this time bound to $o(n \log n)$. One approach is to consider idealized (and non standard) versions of the RAM model; as, for instance, in [KR84] and [PS80], where very large words are assumed. The practicality of such an assumption is unclear. Another approach is to focus on instances of the sorting problem, where the input consists of integers drawn from a restricted interval $[1..m]$. For $m = O(n)$ the known Bucket Sort algorithm applies. It solves the problem in $O(n)$ time. For $m = poly(n)$[1] the variant of the Bucket Sort algorithm, called Radix Sort, runs in $O(n)$ time [Knu73]. More precisely, Radix Sort runs in $O(kn)$ time for $m = n^k$. Thus, a natural extension of the Radix Sort would result with an $o(n \log n)$ time algorithm for $m \leq n^{o(\log n)}$.

[1] We use $poly(n)$ to denote "polynomial in n", and $polylog(n)$ to denote "polynomial in $\log n$".

However, for $m = n^{\Omega(\log n)}$ Radix Sort does not improve on the $O(n \log n)$ time bound.

The second approach was studied for parallel computation as well. Rajasekaran and Reif gave an optimal randomized parallel algorithm in logarithmic time on an arbitrary-CRCW for $m = n \log^c n$, for any constant $c \geq 1$ [RR89]. The integer sorting algorithm of Rajasekaran and Reif cannot be extended for m polynomial in n. For $m =$ [illegible] $n)$, Hagerup provided an $O(\log n)$ time and $O(n^{1+\epsilon})$ space (for any fixed $\epsilon > 0$) parallel algorithm, using $n \log \log n / \log n$ priority-CRCW processors [Hag87]. No optimal parallel algorithm is known for this range. Our sorting algorithm clearly belongs in the second approach.

Using data structures presented by van Emde Boas, Kaas and Zijlstra [vEBKZ77], Johnson [Joh82] dealt with priority queues problems, where the priorities are drawn from the integer domain $[1..m]$. A corollary of his result is an $O(n \log \log m)$ time and $O(m^{1/c})$ space algorithm for sorting, where $c > 0$ is a constant. Johnson recognizes the problem with the space requirements of the algorithm, and writes that the algorithm is not practical and only of theoretical interest.

Kirkpatrick and Reisch [KR84] presented an algorithm, based on a range reduction technique, that has the same complexity bounds as Johnson's algorithm. They state that the algorithm is of little practical value due to both large constants, that are hidden in the asymptotic bounds, and storage requirements.

The following open question, quoted from Kirkpatrick and Reisch ([KR84]), captures an important aspect of our sorting results: "For what ranges of inputs can we construct practical $o(n \log n)$ integer sorting algorithms?". The present paper provides only partial answers to this question, and more work is still needed in order to resolve this question.

The issue of trading space for randomness using random hash functions belongs in the folklore of serial algorithms. For instance, the survey paper [GG88] demonstrates such considerations for hashing large alphabets for string algorithms such as the suffix tree data structure. However, for parallel algorithms this survey mentions only deterministic methods. This immediately suggests an implicit open problem.

Recently, [DadH89] described a dynamic data structure (dictionary) that using randomization supports the instructions *insert*, *delete*, and *lookup*, and that can be implemented in parallel. Time bounds of the form $O(n^\epsilon)$ using $\leq n^{1-\epsilon}$ processors, for some fixed $\epsilon > 0$, are given. However, no time bounds of the form $O(polylog(n))$ are claimed.

Several works have been previously done on relations between PRAM models. The interested reader is referred to the surveys of [EG88] [KR88] [Vis83]. Randomization was previously used in the context of parallel simulations by [KU86] [KRS88] [MV84] [Ran87].

1.2 More on our results

As model of computation for the parallel algorithms, we use mostly the concurrent-read concurrent-write parallel random access machine (CRCW PRAM) family. The members of this family differ by outcome of the event where several processors attempt to write simultaneously into the same shared memory location. In the common-CRCW all these processors must attempt to write the same value (and this value is written). In the arbitrary-CRCW one of the processors succeeds, but we do not know in advance which one. In the priority-CRCW the smallest numbered among the processors succeeds. The above three CRCW models are considered standard. Next we mention two non standard models. In the min-CRCW PRAM the processor that tries to write the minimum value succeeds. In the fetch&add-CRCW PRAM the values are added to the value already written in the shared memory location and all sums obtained in the (virtual) serial process are recorded. Finally, in an exclusive-read exclusive-write (EREW) PRAM simultaneous access of more then one processor into the same shared memory location is not allowed.

A parallel algorithm achieves optimal speed-up if its time$\times$processor product matches the number of operations of the fastest serial algorithm for the problem. Typically, we will state our parallel results in the following form: "x operations and t time". Throughout this paper, this will always translate into "t time using x/t processors". The papers [EG88], [KRS88], [KR88] and [Vis83] overview research directions on parallel algorithms. All of them concede that achieving optimal speed-up, or at least approaching this goal, is a crucial property for parallel algorithms that are intended to be practical. A secondary (but very important) goal is to minimize parallel time. Another critical practical concern is space requirements. These guidelines led us in designing the algorithms of the present paper.

Our (main) randomized sorting algorithms are presented as follows. We first present in Section 2 a deterministic algorithm that takes $O(n \log\log m)$ time and uses $O(m^{1/c})$ space, for any fixed $c \geq 1$. A parallel version of this deterministic algorithm takes $O(\log n)$ time and $O(m^{1/c})$ space, for any fixed $c \geq 1$, using $n \log\log m / \log n$ processors (optimal speedup). This parallel algorithm is designed for the non-standard min-CRCW processors.

The second element in our presentation is described in Section 3. A randomized parallel hashing scheme is presented. It is optimal and takes logarithmic time. Specifically, let W be a given set of n numbers from an arbitrary large domain $[1..m]$. We show how to find a one-to-one mapping $F : W \longrightarrow R$, where $|R| = O(n)$, by a randomized algorithm on the arbitrary-CRCW PRAM. This mapping is computed in $O(\log n)$ expected time, using $\frac{n}{\log n}$ processors. Evaluation of $F(x)$, for each $x \in W$, takes $O(1)$ worst case time. The connection to the sorting results is as follows. We show how to use these hash functions in order to reduce the space requirements in both the serial and parallel algorithms to only $O(n)$ space. The penalty is that now we have randomized algorithms rather than deterministic. For the parallel algorithm, the parallel time increases to $O(\log n \log\log m)$ while, the operation count does not increase (asymptotically) on the min-CRCW model.

The third element in our presentation is described in Section 4. Simulations of the min-CRCW PRAM model by weaker models are presented. Some of the simulations are randomized. Some simulations apply also to the fetch&add-CRCW model.

Denote a CRCW PRAM with a shared memory of size S as *CRCW(S)* PRAM. In the following, we list some upper bounds for simulating one step of an n-processor min-CRCW(S) PRAM:

- $O(\log n)$ expected time on an $\frac{n}{\log n}$-processor arbitrary-CRCW($S + O(n)$) PRAM (optimal speedup).
- $O(\log n)$ time on an $\frac{n \log\log n}{\log n}$-processor priority-CRCW($O(S + n^{1+\epsilon})$) PRAM ($\epsilon > 0$).
- $O(\log\log m)$ time on an n-processor arbitrary-CRCW($O(mS)$) PRAM, where m is an upper bound for the value that can be written to a memory cell by the min-CRCW PRAM.

The first result is an improvement over a previously known result [EG88] where there was a restriction that the memory addresses being used by the simulated min-CRCW are of at most $O(\log n)$-bit size. The result can be extended to simulating one step of a fetch&add-CRCW PRAM. We are not aware of similar (i.e., optimal simulation) results even for simulation of the (relatively weaker) priority-CRCW PRAM by an arbitrary-CRCW PRAM. The last two simulations are deterministic.

Combining the optimal simulation from Section 4, the parallel hashing scheme in Section 3 and the algorithm in Section 2 we derive a randomized parallel algorithm for sorting n integers from the range $[1..m]$ on an arbitrary-CRCW that achieves $O(\log n \log\log m)$ expected time, $O(n \log\log m)$ expected number of operations and $O(n)$ space.

Not only the space efficient parallel sorting result can benefit from the simulations. Recall the original min-CRCW PRAM sorting algorithm of Sec. 2. Together with some of these simulations, we get the following deterministic sorting results on standard PRAM models: (1) $O(\log n \log\log m)$ time and $O(m^\epsilon)$ space (with any fixed $\epsilon > 0$) using $\frac{n \log\log n}{\log n}$ priority-CRCW processors; and (2) $O(\log n)$ time and $O(m^\epsilon)$ space ($\epsilon > 0$) using $\frac{n(\log\log m)^2}{\log n}$ arbitrary-CRCW processors.

Some of the ideas we use in the deterministic algorithms of Section 2 go back to [vEBKZ77]. These ideas were inspired also by the algorithms of [Hag87] and [Joh82]. Johnson's algorithm has the same complexity as our deterministic serial algorithm. However, our sorting algorithm has two advantages: it is simpler and parallelizable.

The proposed parallel hashing scheme may be a useful tool for parallel algorithms that use large space. We demonstrate this with several algorithms (in addition to our sorting algorithm) for which the space requirement is large. By using the proposed parallel hashing scheme, they become efficient and possibly practical.

There are applications that relate to combinatorial algorithms on strings. If the alphabet is large then a naming assignment procedure for substrings is essential to avoid large space. Such deterministic procedure, due to [GG88], takes $O(n \log n)$ operations and evaluation of a name takes $O(\log n)$ time. (We actually referred earlier to the same procedure.) Using our parallel hashing scheme, the naming assignment takes $O(n)$ expected number of operations and evaluation of a name takes $O(1)$ time in the worst case.

Another application is in the construction of *suffix trees*. Known parallel algorithms for this problem require $O(n^{1+\epsilon})$ space [AIL+88] or $O(n \log n + m^2)$ space [GG88] where n and m are the lengths of the text string and the pattern string respectively. Space requirements in both algorithms can be reduced to $O(n)$ and $O(n \log n)$, respectively, in exchange for randomization and an $O(\log n)$ increase in time (but no change in number of operations).

The parallel hashing scheme may be used to reduce the space in Hagerup's sorting algorithm [Hag87] from $O(n^{1+\epsilon})$ to $O(n)$ in exchange for randomization and an $O(\log \log n)$ increase in time (but no change in number of operations). The parallel hashing scheme is also used in the optimal simulation in Section 4.

Recall that we use the non standard min-CRCW model in Section 2. Note that we do not advocate this model as an alternative for existing "acceptable" theoretical models for parallel computation. The methodological attitude here is: (1) design the algorithm on the min-CRCW model, and (2) show later how to simulate this model on more acceptable ones.

Postscript. After all results in the present paper have been achieved and a first draft has been distributed, we found out that very recently Bhatt et al. [BDH+89] designed independently a parallel deterministic integer sorting algorithm which is related (though, not identical) to the basic construction of Section 2.1 below. Using a new list ranking algorithm they were even able to reduce the time to $O(\log n / \log \log n + \log \log m)$ maintaining optimal speed up. However, none of the randomized and only part of the deterministic space reductions ideas of the present paper appear there. Finally, we note that we do not see how to use their results for further improving our randomized results. This is since the list ranking part is not the main bottleneck for improving our space-efficient randomized sorting algorithm.

2 The Deterministic Algorithm

We consider the following problem:

Input: Sequence $x[1], \ldots, x[n]$ of distinct integers drawn from the domain $[1..m]$, for some integer m. **Problem:** Sort the sequence. (Formally, compute the permutation π of $\{1,...,n\}$ such that $x[\pi(1)],...,x[\pi(n)]$ is sorted in non-decreasing order.)

In Section 2.4 we discuss ways for withdrawing the distinction assumption.

For presentation purposes, we falsely assume that the sequence $x[1],...,x[n]$ is given in the following redundant form. There is a *domain array* of bits $D[1..m]$ so that $D[i] = 1$ if the value of some element $x[j]$ is i and $D[i] = 0$ otherwise. Given bit $D[i] = 1$ we define the smallest $i_1 > i$ such that $D[i_1] = 1$ to be the *right neighbor* of $x[i]$.

This neighborhood relation translates easily to the values of the input sequence: the *domain right neighbor* $drn[i]$ of $x[i]$ is the element $x[j] = \min\{x[k] : x[k] > x[i]\}$. The array drn defines a linked list, where the elements preceding $x[i]$ in the linked list are smaller than $x[i]$ and the elements succeeding $x[i]$ are larger. The distance from the beginning of the list is the *rank* of $x[i]$ relative to the input elements.

We solve the sorting problem in two steps:

(a) Compute the *domain right neighbor* of each index i.

(b) For each element $x[i]$, compute its rank r in the linked list defined by the drn, and let $\pi(r)$ be i.

Step (b) can be (trivially) done in $O(n)$ time or in parallel time $O(\log n)$ and optimal speed up using a the List Ranking algorithm ([AM88]), [CV86], [CV88], [CV89]). Therefore, our main concern is solving the domain right neighbor problem. For simplicity we assume that $m = 2^{2^t}$ for some integer $t \geq 1$. The domain right neighbor, as defined above, is the nearest neighbor from the right. We will also need a definition of the *domain left neighbor*, $dln[i]$, of element $x[i]$: the element $x[j] = \max\{x[k] : x[k] < x[i]\}$.

2.1 Algorithm for finding Domain Nearest Neighbors

The algorithm is recursive. The main effort is in defining *precisely* the problem that is being solved recursively. The recursive algorithm will provide solutions for the problems of finding left and right domain nearest neighbors. For each element the recursive algorithm separately treats the domain right neighbor and the domain left neighbor computations. This is done by duplicating each element $x[i]$ into a *left copy* $x_l[i]$ and a *right copy* $x_r[i]$. Intuitively, copy $x_r[i]$ is "responsible" for finding the domain right neighbor and $x_l[i]$ is "responsible" for finding the domain left neighbor. Initially, $x_l[i] = x[i]$ and $x_r[i] = x[i]$.

In addition, two auxiliary copies are added $x_r[n+1]$ and $x_l[n+1]$. Copy $x_r[n+1] = 1$ is the domain left neighbor of the smallest input element. Similarly, copy $x_l[n+1] = m$ is the domain right neighbor of the largest input element. (Note that at this stage only for $i = n+1$, $x_r[i]$ is not equal $x_l[i]$.) We assume, without loss of generality, that the input elements are from $[2..m-1]$.

Informally, our algorithm works as follows: The input elements are from an interval I. Interval I is partitioned into small intervals, defining local problems of domain nearest neighbors searches. For each subinterval I_k, at most two elements (smallest left copy and largest right copy) might not have their neighbors in I_k. Such elements are collected into the global sets GR and GL. Thus, a problem on interval I is reduced recursively into several local problems on subintervals I_k and one global problem. By choosing I_k to be of size $\sqrt{|I|}$ (for all k), we have that all local problems and the global problem are with intervals of size $\sqrt{|I|}$.

The input for the recursive algorithm includes two sets L and R, whose values belong to an interval I of integers. I is of the form $a+[1..m']$, i.e., $I = \{a+1, a+2, ..., a+m'\}$ for some a and m'. Set L will always represent a non-empty subset of the left copies and set R will always represent a non-empty subset of the right copies. Each element in set L searches for its *left neighbor* within set R. This left neighbor is defined as the largest element in R which is smaller than it. Similarly, each element in set R searches for its *right neighbor* defined as the smallest element in L which is larger than it.

Initially, the interval I is $[1..m]$ (i.e., $a = 0$ and $m' = m$), L is $\{x_l[1], ..., x_l[n+1]\}$ and R is $\{x_r[1], ..., x_r[n+1]\}$.

The recursive algorithm DNN

Input: L, R and I, where L and R are nonempty sets and $I = a+[1..m]$ is an interval of integers. We refer to $m = |I|$ as the *size* of the problem.

A processor stands by each element in L and each element in R.

if $m = 2$ (*comment:* the situation for a recursive problem for which $m = 2$ is characterized in Corollary 2)

then Declare the element of L to be the right neighbor of the element of R and the element of R to be the left neighbor of the element of L.

else

(1) Partition set L into $q = \sqrt{m}$ subsets $L_0, L_1, ..., L_{q-1}$ where L_k contains elements (of L) from the interval $I_k = a + k \cdot q + [1..q]$, for $k = 0, ..., q-1$. Similarly, partition set R into $q = \sqrt{m}$ subsets $R_0, R_1, ..., R_{q-1}$ where R_k contains elements (of R) from the interval $I_k = a + k \cdot q + [1..q]$, for $k = 0, ..., q-1$.

(2) Let a_k be the smallest element in L_k. **If** a_k is less than or equal to the smallest element in R_k **then**: (a) Select the integer k into the new set GL (integer k represents element

a_k and will be referred to as a_k). (b) Remove a_k from L_k.

Similarly, let b_k be the largest element in R_k. **If** b_k is greater than or equal to the largest element in L_k **then**: (a) Select the integer k into the new set GR (integer k represents element b_k and will be referred to as b_k). (b) Remove b_k from R_k.

(3) Do the following recursive calls: (a) For each pair of nonempty (local) subsets L_k and R_k solve the problem for an input consisting of L_k, R_k and I_k. (b) If (global) sets GL and GR are nonempty then solve the problem for an input consisting of GL, GR and $J = a + [0..q-1]$.

Comments:

- If the smallest (resp. largest) element of $L \bigcup R$ in I_k is in L (resp. in R) then its left (resp. right) neighbor is not in I_k. We collect the elements from L (resp. from R) whose left (resp. right) neighbor is not in I_k into the global set GL (resp. set GR).
- The algorithm advances to the deepest level of recursion and simply terminates (without any backtracking).
- All recursive calls of the algorithm are performed simultaneously in parallel.

2.2 Correctness

Proposition 1 *Let $x[j]$ be the left neighbor of $x[i]$. Then for each level of the recursion the following properties hold:*

(a) The values of all elements in L are distinct and the values of all elements in R are distinct.

(b) $x_l[i]$ and $x_r[j]$ are both represented in the same recursive sub-problem.

(c) $x_r[j] < x_l[i]$.

(d) $x_r[j]$ is the left neighbor of $x_l[i]$.

(e) Each element is represented in exactly one recursive sub-problem.

(f) Set L is nonempty if and only if set R is nonempty.

The proof of Proposition 1 is given in Appendix A.

Corollary 2 *If a problem with input L, R and I is of size 2 then $|L| = |R| = 1$ and the element in R is the left neighbor of the element in L.*

Proof: Assume that $x_l[i] \in L$ and that $x[j]$ is the left neighbor of $x[i]$. Let $I = [a, a+1]$ for some integer a. Following from property (b) in Proposition 1, $x_r[j] \in R$. Thus, from property (c) we have $x_r[j] = a$ and $x_l[i] = a+1$. If $L > 1$ then by property (a) a second element can only be $x_l[i'] = a$ (for some $i' \neq i$). However, following property (c) its left neighbor must be $< a$, contradicting property (b). Similarly, R may contain only one element. ∎

Corollary 3 *Suppose that initially m, the size of the interval from which the input elements are drawn, is 2^{2^t} for some integer $t \geq 1$ (and $t = \log\log m$). Consider a recursive problem at recursion level t. The interval I from which the input elements for such recursive problem are drawn consists of two successive integers. That is, for some integer x, $I = x + [1..2]$. Furthermore, $L = \{x+2\}$ and $R = \{x+1\}$ and $x+2$ is the right neighbor of $x+1$.*

2.3 Complexity and implementation

We start by discussing the complexity of algorithm DNN and later state the sorting results.

Lemma 4 *Given are n elements with distinct values from the interval of integers* $[1..m]$. *(1) Algorithm DNN works serially in $O(n \log\log m)$ time and $O(m)$ space. (2) Algorithm DNN works in $O(\log\log m)$ time and $O(m)$ space, using n processors on a min-CRCW PRAM.*

Proof: The size of interval I for each recursive sub-problem is bounded by $\sqrt{m}$. Therefore, $D(m)$, the depth of the recursion, satisfies $D(m) \leq D(\sqrt{m}) + O(1)$, implying $D(m) = O(\log\log m)$. For each level of the recursion we need to perform at most $O(n)$ operations and therefore the total number of operations is $O(n \log\log m)$.

To finish deriving the serial result we "remember" for each element in each level of the recursion its original index in the initial sets L and R. The space needed for all subproblems together in each level of the recursion is $O(m)$. Since we can reuse the space for the highest level of the recursion, we get a total of $O(m)$ space, as well. Item (1) of the lemma follows.

We proceed to the parallel result. Initially, we assign one processor to each copy of each element (i.e., each element of the original sets L and R). This assignment remains throughout the entire algorithm.

For step (2), we need to have in a_k (resp. b_k), for $k = 0, .., q-1$, the minimum (resp. maximum) element's value over all the elements that belong to L_k (resp. R_k). We already argued (implicitly item (1) of the lemma) that sequentially this can be trivially done in $O(n)$ time. In parallel, step 2 can be done in $O(1)$ time, using n processors. Here is where we take advantage of the min-CRCW PRAM, where if several processors try to write into the same memory location, only the one with the minimal value succeeds. Item (2) of Lemma 4 follows. ∎

2.4 Extensions

Withdrawing the distinctness assumption.

We assumed that all elements in the input sequence are distinct. The purpose of this subsection is to extend algorithm DNN and show that this assumption is not necessary. It is trivial to achieve distinctness by replacing each input element $x[i]$ by the pair $< x[i], i >$. The problem is that this enlarges the required space to $O(nm)$ Fortunately, this will not cause any problem in deriving the randomized sorting results in Section 4.1.

However, the problem remains if we are interested in deterministic results. The remainder of this subsection is devoted to giving alternative extensions that are less space consuming.

We allocate processor i to element i, for each $1 \leq i \leq n$. For each element i, let $s(i)$ be the smallest index of an element equal to element i (formally $x[i] = x[s(i)]$ and for $j < s(i)$, $x[i] \neq x[j]$). We make use of a *bulletin board* $BB[1..m]$ (this term was previously used by Galil [Gal84]).

Step 1. Processor i writes i into location $BB[x[i]]$. Since the min-CRCW PRAM is used location $BB[x[i]]$ will contain $s(i)$. Step 1 uses $O(n)$ space.

Only processors (and their respective elements) that succeed in writing their index participate in Step 2. Formally, these are all elements i for which $s(i) = i$. All elements participating in Step 2 are distinct.

Step 2. Apply the DNN algorithm. As a result we get the domain right neighbor relative to all elements participating in Step 2.

Step 3. Associate the pair $< s(i), i >$ with element i, for each i, and impose a lexicographic order on these pairs. Apply the DNN algorithm with respect to these pairs. (It is easy to map these pairs into the domain of integers $[1..n^2]$, preserving the lexicographic order. So Step 3 needs $O(n^2)$ space.)

Step 4 derives the desired right neighbor relation from the outcome of Steps 2 and 3. Let $x[k]$ be the right neighbor of $x[s(i)]$, as a result of Step 2, and let $< s(j), j >$ be the right neighbor of the pair $< s(i), i >$, as a result of Step 3.

Step 4.

```
if $s(i) = s(j)$
then $x[j]$ is the right neighbor of $x[i]$
else $x[k]$ is the right neighbor of $x[i]$.
```

We conclude,

Lemma 5 *Given are n elements with values from the interval of integers $[1..m]$. (1) The extension of algorithm DNN works serially in $O(n \log\log m)$ time and $O(m+n^2)$ space. (2) The extension of algorithm DNN works in $O(\log\log m)$ time and $O(m+n^2)$ space, using n processors on a min-CRCW PRAM.*

Reducing (deterministically) the space requirements.

For simplicity, let us assume that $m > n^2$. The reason is that our results extend known results mostly when this assumption holds.

Lemma 6 *Given are n elements with values from the interval of integers $[1..m]$. Algorithm DNN can be further enhanced to: (1) run serially in $O(n \log\log m)$ time and $O(m^{\frac{1}{c}})$ space, where $c \geq 1$ is any fixed constant. (2) run in $O(\log\log m)$ time and $O(m^{\frac{1}{c}})$ space (for any constant $c \geq 1$), using n processors on a min-CRCW PRAM.*

Proof: The output of the extended DNN algorithm defines a linked list of the elements defined by the right neighbor relation. This linked list is *stable* in the following sense: if $x[i] = x[j]$ for some $i < j$ then element i precedes element j in the linked list. A consequence is that an iterative method, in the spirit of Radix Sort, can be applied. Thus, given an algorithm of time T and space S, for each integer $c > 0$, we can have a DNN algorithm with $O(cT)$ time and $O(S^{1/c})$ space. ∎

Our primary concern in this section is the sorting problem. So, if we add the missing list ranking step to the DNN algorithm we get,

Theorem 1 *Given are n elements with values from the interval of integers $[1..m]$, our sorting algorithms achieve the following results: (1) $O(n \log\log m)$ serial time and $O(m^{\frac{1}{c}})$ space, where $c \geq 1$ is any fixed constant. (2) $O(\log n)$ parallel time and $O(m^{\frac{1}{c}})$ space (for any constant $c \geq 1$), using $\frac{n \log\log m}{\log n}$ processors on a min-CRCW PRAM.*

If the input is from a range polynomial in n then we have the same complexities as in Hagerup's algorithm [Hag87]. The range for which our algorithm gives better results than the best known algorithms is for $n^{\log\log n} \ll m \ll 2^{n^{\alpha(1)}}$, where $\ll$ denotes smaller asymptotically. Thus, for example, for $m = n^{polylog(n)}$ we have:

Corollary 7 *n integers from the range $[1..n^{\log^k n}]$ (for every constant $k > 0$) can be sorted in: (1) $O(n \log\log n)$ serial time and $O(n^{\frac{\log^k n}{c}})$ space, for any constant $c \geq 1$; and (2) $O(\log n)$ parallel time and $O(n^{\frac{\log^k n}{c}})$ space, for any constant $c \geq 1$, using $\frac{n \log\log n}{\log n}$ processors on a min-CRCW PRAM.*

3 Trading Space for Randomness

In this section we show how to use randomization in order to reduce the space complexity of the algorithms to $O(n)$. The randomization is of the "Las-Vegas" type algorithm. That is, some of the steps of the space reducing algorithm are based on randomized moves and it never errs.

Recall that in each recursive level of algorithm DNN, there are $O(m)$ variables a_k and b_k. However, in each level of the algorithm only $O(n)$ of these variables are actually used. For a given level denote by W the set of a_k and b_k variables that are being used ($|W| = O(n)$). The deterministic implementation in Sec. 2 requires $O(m)$ space for the a_k and b_k variables. The key idea here is to randomly select a hash function for mapping the set W into $O(n)$ space. In order to avoid collisions, we shall use a *perfect hash* function (which is a one-to-one function).

Remark. For the serial algorithm, much simpler hash functions would suffice in order to reduce the space to $O(n)$. From now on we concentrate on the parallel algorithm. Based on this, the implementation for the serial algorithm will be straightforward.

A basic procedure for constructing a perfect hash function is presented. Its expected running time is logarithmic and its expected number of operations is linear. Specifically, we prove in Subsection 3.1 the following theorem:

Theorem 2 *Let W be a set of n numbers from the range $[1..m]$, where $m+1 = p$ is prime. Suppose we have $\frac{n}{\log n}$ processors on an arbitrary-CRCW PRAM. A one-to-one function $F : W \rightarrow [1..5n]$ can be found in $O(\log n)$ expected time. The evaluation of $F(x)$, for each $x \in W$, takes $O(1)$ arithmetic operations with numbers from $[1..m]$.*

We show later that Theorem 2 leads to the following:

Corollary 8 *A. Algorithm DNN (and its extension) takes $O(\log n \log\log m)$ expected time and $O(n)$ space, using $\frac{n}{\log n}$ processors on a min-CRCW PRAM. B. The same performance is obtained for the problem of sorting n integers drawn from a domain of size m.*

3.1 Constructing a parallel perfect hash function

In this subsection we prove Theorem 2.

Given is a set W of n numbers from the range $[1..m]$, where $p = m+1$ is prime. The hash function F maps W into the range $[1..5n]$. We use the fundamental perfect hash function

F that was suggested by Fredman, Komlós and Szemerédi [FKS84], as described below. An efficient parallel construction of F is then presented.

Define $f_k : W \to [1..n]$ as $f_k(x) = 1 + (kx \bmod p) \bmod n$ where k is a parameter from $[1..m]$. Let $B(k,j)$ be the set of values in W that are mapped by f_k into j, i.e. $B(k,j) = \{x \in W : f_k(x) = j\}$. Also, let $b(k,j) = |B(k,j)|$ and $S_k = \sum_{j=1}^{n} b(k,j)^2$. For each $j = 1,\ldots,n$, define $f'_{k',r} : B(k,j) \to [1..r^2]$ as $f'_{k',r}(x) = 1 + (k'x \bmod p) \bmod r^2$ where $k' = k'(j)$ is a parameter from $[1..m]$ and $r = b(k,j)$. Construction of the hash function F will be based on two steps:

(a) Select k for which $S_k < 5n$.

(b) For each j, select $k'(= k'(j))$ for which $f'_{k',b(k,j)}$ is one-to-one.

After all parameters k and $k'(j)$'s (for all $j = 1,...,n$) are appropriately selected, the one-to-one function F is derived by first applying f_k and then $f_{k'(j),b(k,j)}$ for a proper j. Specifically, F is constructed as follows. For each set of elements $B(k,j)$, $b(k,j)^2$ space is assigned. Let M_i be the prefix sum $\sum_{j=1}^{i} b(k,j)^2$. Then $[1..M_i]$ is an array assigned to the first i sets $B(k,1),...,B(k,i)$. The function f_k maps each element in $B(k,j)$ into $[(M_{j-1}+1)..M_j]$. $F(x)$ is evaluated as follows:

(a') Evaluate $j = f_k(x)$.

(b') $F(x) = M_{j-1} + f'_{k',b(k,j)}(x)$.

Step (a) guarantees that the overall space $M_n = S_k$ is linear ($< 5n$). Step (b) guarantees that the mapping is one-to-one. It remains to show how to implement steps (a) and (b).

Fact 1 ([FKS84]) For at least one-half of the values k in $[1..m]$, $S_k < 5n$. Thus, for a randomly selected k, $S_k < 5n$ with probability $\geq \frac{1}{2}$.

Define $k' = (k'(j))$ to be *good* if $f'_{k'(j),b(k,j)}$ is a one-to-one function (over $B(k,j)$).

Fact 2 ([FKS84]) For each j in $[1..n]$, at least one-half of the k's in the range $[1..m]$ are good.

To construct F, we apply the following randomized procedure:

(a) Repeatedly select k at random until $S_k < 5n$.

(b) For each j, repeatedly select $k' = k'(j)$ at random until it is good.

Implementation of Step (a).

Following fact 1, the expected number of iterations in (a) is ≤ 2. We first show how to check whether $S_k < 5n$. Given $b(k,j)$ for all j, the evaluation of the prefix sums $M_i = \sum_{j=1}^{i} b(k,j)^2$ (for $i = 1, ..., n$) and of $S_k = M_n$ can be done by using the Prefix Sums algorithm of Cole and Vishkin [CV86] [CV89] in $O(\log n / \log\log n)$ time and $O(n)$ operations. The evaluation of $b(k,j)$ for each j is done as follows:

(1) Sort the n numbers in $\{f_k(x) : x \in W\}$ into an array $C[1..n]$.

(2) Find for each j the rightmost (resp. leftmost) index i_1 (resp. i_2) for which $C[i_1] = j$ (resp. $C[i_2] = j$). Let $b(k,j)$ be $i_1 - i_2 + 1$.

Step (2) can be trivially done in $O(1)$ time using n processors. To do step (1), note first that the range of f_k is the integer interval $[1..n]$. Thus, we may employ the integer sorting algorithm due to Rajasekaran and Reif:

Lemma 9 *([RR89]) n keys from the range $[1..n]$ can be sorted using $\frac{n}{\log n}$ arbitrary-CRCW PRAM processors in $O(\log n)$ time, with probability $\geq 1 - \frac{1}{n^\alpha}$, for any constant $\alpha > 0$.*

Following the above we have that each iteration in step (a) of the construction of F takes $O(n)$ expected number of operations and logarithmic expected time. We conclude

Lemma 10 *Given is a set W of n numbers from the range $[1..m]$ and some $k \in [1..m]$. Checking whether $S_k < 5n$ can be done in $O(\log n)$ expected time, using $\frac{n}{\log n}$ arbitrary-CRCW processors.*

Corollary 11 *Step (a) in the construction of F takes $O(\log n)$ expected time, using $\frac{n}{\log n}$ arbitrary-CRCW processors.*

Implementation of Step (b).

In step (b) the procedure to check whether k' is good for j is easy when using the arbitrary-CRCW PRAM. Our goal is to select a good k' for each j within a total of $O(n)$ operations and logarithmic time. The difficulty is that Step (b) should be done independently for each j ($j = 1, ..., n$). We prove

Lemma 12 *A good $k' = k'(j)$ can be found for all j, $j = 1, .., n$, in $O(\log n)$ expected time, using $\frac{n}{\log n}$ processors.*

Proof: To prove the lemma we show three facts: (I) the expected maximum number of trials in selecting a good k' for j (over $j = 1,..,n$) is $O(\log n)$; (II) the expected total work of selecting good k's for all j ($j = 1,..,n$) is $O(n)$; and (III) the processors can be allocated according to (a) and (b) to yield an $O(\log n)$ expected time parallel procedure for step (b) in the construction of F, using $\frac{n}{\log n}$ processors.

Let t_j be the number of *trials* before a good k' is found for j, and let $t = max\{t_j : j = 1,\ldots,n\}$.

Claim 13 $E[t] \leq 2(\lceil \log n \rceil + 1)$.

Proof: Let N_i be the number of js for which a good k' was not found in the first i trials; i.e. $N_i = |\{j : t_j > i\}|$. Consider the following Bernoulli trials: A *success* in the i'th trial is defined to be the case that the number of good k's found in the i'th trial is at least $\frac{N_i}{2}$; that is, a success is when $N_{i+1} \leq \frac{N_i}{2}$. Following fact 2, $Prob[success] \geq \frac{1}{2}$.

Let x be the number of trials until the $(\lceil \log n \rceil + 1)$'th successful trial. It is easy to see that t is bounded by x and therefore $E[t] \leq E[x] \leq 2(\lceil \log n \rceil + 1)$. ∎

Claim 13 proves fact (I). To prove fact (II) we first show

Claim 14 $E[t_j] \leq 2$ *for each* j.

Proof:

$$E[t_j] = \sum_{i=1}^{\infty} i \cdot Prob[t_j = i] \leq \sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=1}^{\infty} \sum_{k=1}^{i} \frac{1}{2^i}$$

changing the order of summation, we get

$$= \sum_{i=1}^{\infty} \sum_{k=i}^{\infty} \frac{1}{2^k} = \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} = 2.$$

∎

Let op_j be the number of operations required for selecting a good k' for j. Since $op_j = t_j b(k,j)$ we have $E[op_j] = E[t_j]b(k,j) \leq 2b(k,j)$ and the total number of operations is expected to be $E[\sum_{j=1}^{n} op_j] \leq 2\sum_{j=1}^{n} b(k,j) = 2n$.

It remains to give an implementation for the processors allocation. The selection of the numbers $k'(j)$ is done in phases. In each phase a new number k' is selected and tested, for each j such that no good $k'(j)$ has been found. An element $x \in B(k,j)$ is called *active* if no good $k'(j)$ has yet been found (for j). Let N_i' be the number of active elements in phase i.

We use a standard idea. Initially, N'_1 is n. As long as $N'_i > n/\log n$, we simply compact all N'_i active elements of phase i into an array of length N'_i prior to the phase. Consider the first phase j for which $N'_j \leq n/\log n$. The compacted array of size N'_j, will be used for all subsequent phases.

Finally, a trivial application of Brent's scheduling theorem will provide actual assignment of processors to jobs.

Specifically, let $VAL[1..5n]$ be an array. The perfect hash function F, being constructed, will map each element of the input set W into array VAL.

Less informal implementation of Step (b).

```
$i := 1$; $N'_i := n$.

While $N'_i > n/\log n$ do
    (All active elements in $W$ are in array $ACTIVE[1..N'_i]$ sorted by the $B(k,j)$ to which
    they belong)

    Phase $i$
    1. The first element in an active $B(k,j)$ selects at random a $k'(j)$ value.
    2. Each active element $x$ evaluates its hashing value $F(x)$ using $f_{k'(j)}$ and writes its
       original value $x$ into $VAL[F(x)]$ (using the arbitrary CRCW convention).
    3. Each active element $x$ checks whether its value is written in $VAL[F(x)]$. If not it
       "disqualifies" the $k'(j)$ for its $B(k,j)$ set.
    4. All active elements belonging to $B(k,j)$ sets whose $k'(j)$ was disqualified remain
       active in phase number $i+1$. Their number is $N'_{i+1}$.
    5. Using a prefix sums algorithm, compact them into array $ACTIVE[1..N'_{i+1}]$
    6. $i := i+1$.

end while

Denote $j = i$. $N'_j$ is at most $n/\log n$.

While there is $j$ for which good $k'(j)$ has not been found do
    (All active elements in $W$ are in array $ACTIVE[1..N'_j]$, sorted by the $B(k,j)$ to which
    they belong)

    Do steps 1-4 and 6 as above.

end while
```

Complexity Following fact 2, $E[N_i'] \le \frac{n}{2^{i-1}}$. Therefore, the expected total work is $\sum_i E[N_i'] \le 2n$. The time in each phase in part 1 is dominated by the compaction procedure (step 5). Using the Prefix Sums algorithm each phase takes $O(\frac{\log n}{\log\log n})$ time. Using arguments similar to those in Claim 13, $E[j] \le 2\log\log n$ where j is the number of phases in the first part. The expected number of phases for part 2 is $O(\log n)$ based on Claim 13.

Using Brent's theorem [Bre74] step (b) can be implemented in $O(\log n)$ expected time, using $\frac{n}{\log n}$ processors on an arbitrary-CRCW PRAM. ∎

Having Lemma 10 and Lemma 12, Theorem 2 immediately follows.

3.2 Applications

As a motivation for the previous subsection we stated Corollary 8.

Proof of Corollary 8. In algorithm DNN (and its extension) there are $\log\log m$ phases. Separately for each phase we hash the $O(n)$ variables a_k and b_k. Part A of the Corollary 8 follows. Part B is trivial. ∎

We mention here some examples of algorithms for which the parallel hashing scheme can be used.

One application relates to the construction of *suffix trees*. This is probably the most important data structure for algorithms on strings. Applications of this data structure are reviewed in [Apo84]. [GG88] indicate that the space requirement of the suffix tree construction is the source of inefficiency in quite a few parallel preprocessing algorithms. The parallel algorithm by [AIL+88] for constructing suffix tree requires $O(\log n)$ time, $O(n\log n)$ operations and $O(n^{1+\epsilon})$ space (for any $0 < \epsilon \le 1$), where n is the length of the input string. Using the parallel hashing scheme, the space requirement decreases to $O(n)$, while the time increases to expected $O(\log^2 n)$ and the number of operations remains $O(n\log n)$ (as expected value rather than worst case). Suppose we have a relatively short string, whose length is m, and a long string, whose length is n. [GG88] considered instances where suffix trees are needed only for supporting queries requesting comparison among substrings of the short string and the long string. Their algorithm takes $O(\log m)$ time, $O(n\log m)$ operations and $O(n\log n + m^2)$ space. Using the parallel hashing scheme, the space requirement decreases to $O(n\log n)$, the time increases to expected $O(\log n \log m)$ and the (expected) number of operations remains $O(n\log m)$.

Section 2 in the approximate string matching survey of [GG88] discusses various ways for hashing many different substrings of a certain string. This is a fundamental problem that arises in some string matching automaton-like algorithms. They consider both serial and parallel computation. Given a string x, the size of the alphabet is relevant to the assignment of names to different substrings of x. In principle, different names should be assigned to different substrings of a given string x. In cases where the number of different substrings is n, a name should be a number from $[1..n]$. Name assignment is a mapping from a possibly large domain into $[1..n]$. Galil and Giancarlo propose an assignment procedure that takes $O(n \log n)$ operations where $|x| = n$ [GG88]; subsequently, it takes $O(\log n)$ time to find the name of a substring. Using our parallel hashing scheme, the assignment procedure takes $O(n)$ expected number of operations and $O(\log n)$ expected time; finding a name for a given substring takes $O(1)$ worst-case (!) time.

Lamdan and Wolfson [LW88] use hashing for object recognition. Their method is parallel in a straightforward manner, except for their hash table construction. The parallel hashing scheme can be useful there.

Hagerup's algorithm for sorting integers from polynomial range [Hag87] has the drawback of using $O(n^{1+\epsilon})$ space (for any fixed $\epsilon > 0$). By using the parallel hashing scheme its space complexity decreases to $O(n)$. The expected number of operations remains the same and the time increases from $O(\log n)$ to expected $O(\log n \log \log n)$.

Finally, the parallel hashing scheme is used to get an optimal randomized simulation of the min-CRCW PRAM by arbitrary-CRCW PRAM, as given in Section 4.

Comment on finding a prime in a given range.

We assumed above that $m + 1$ is a prime. To withdraw this assumption we should give a procedure that, given some m, finds a prime $p > m$ such that $\log \log p = O(\log \log m)$; i.e. $p \in [(m+1)..m^{\log^k m}]$, for some constant $k > 0$. We have some preliminary results on this. In particular, we know how to find p in $O(n)$ operations. The parallel time complexity is $O(\log^3 m)$ with high probability. To see the significance of such a procedure we should refer to the way in which the sorting algorithm is viewed. If the algorithm is for a fixed range $[1..m]$ then finding p is just a preprocessing which may be done only once (p can then be part of the algorithm's description). We may, however, use the sorting algorithm as an *input sensitive* algorithm with no a priori knowledge about the range. Specifically, after reading the input values the actual range may be found by using a maximum finding procedure. In this case, an efficient procedure for finding a prime p is desired.

4 Simulating the min-CRCW PRAM

In this section we deal with simulations of the min-CRCW PRAM by weaker (and more acceptable) models of parallel computation. We show applications of some of these simulations for the parallel Sorting algorithms.

Our most interesting simulation result is the following:

Simulation result 1 One step of an n-processor min-CRCW PRAM can be simulated by an $\frac{n}{\log n}$-processor arbitrary-CRCW PRAM in $O(\log n)$ expected time (optimal speed up) and $O(n)$ additional space.

Before proceeding to prove this simulation result, we make some general comments on how the result should be read and what has to be proved. These comments apply to other simulation results below, as well. The difference between the min-CRCW PRAM, being simulated, and the simulating arbitrary-CRCW PRAM lies in the way write conflicts are resolved. For this reason our proof need to be concerned only with a 'write' stage of the min-CRCW PRAM on the arbitrary-CRCW PRAM. The space requirements for the simulating arbitrary-CRCW PRAM be read as follows: (1) it needs as much space as the min-CRCW PRAM; in addition, (2) $O(n)$ space is needed.

Lemma 15 *Consider the problem of simulating a single 'write' stage of an n-processor min-CRCW PRAM on an arbitrary-CRCW PRAM. This problem can be reduced in $O(\log \log n)$ time and $O(n)$ operations (on an arbitrary-CRCW PRAM), to the problem of Sorting n integers from the range $[1..n]$. The reduction uses $O(m)$ space, where m is the size of the memory in the simulated min-CRCW PRAM.*

Proof: Suppose the memory of the min-CRCW PRAM is an array $M[1..m]$ of size m. We denote the processors of the simulated min-CRCW PRAM by $MP_i, 1 \leq i \leq n$. As usual we will refer to the computation on the simulating arbitrary-CRCW PRAM in terms of operations, and suppress the issue of allocation of these operations to processors of the simulating machine. We will make one exception to this, in a case where such allocation requires special care. A typical 'write' stage of n min-CRCW PRAM processors can be viewed as follows. Processor $MP_i, 1 \leq i \leq n$, attempts to write value v_i into target address $M[t_i]$. Let S_i be the set of elements j such that $t_j = t_i$. The definition of the min-CRCW PRAM implies that v_i is written into $M[t_i]$ if $v_i = \min\{v_j : j \in S_i\}$.

The simulation makes use of a *bulletin board* $BB[1..m]$ that enables direct communication between all elements with the same target address. It works as follows:

a. For each processor MP_i, write its index i into memory location $BB[t_i]$. Arbitrarily, some index $i' \in S_i$, succeeds and i' is written into $BB[t_i]$.

The main idea is to "label" each processor in S_i by the same label i' and group all processors in S_i together into a successive subarray (Step (b)). The simulation of the min-CRCW PRAM is carried out by determining the minimum value v_i over each such successive subarray (Step (c)).

b. $LABEL[i] \leftarrow BB[t_i]$. Sort array $LABEL$ into an array $G[1..n]$.

Identical $LABEL$ values occupy successive subarrays of G. The beginning and end of each such subarray can be easily determined.

c. For each such successive subarray in G, find the minimum v_i over $\{v_j : LABEL[j]$ is in the subarray$\}$, and write v_i into memory location $M[t_i]$.

Step (a) can be done in $O(1)$ time and $O(n)$ operations using the arbitrary-CRCW PRAM. Step (c) can be done in time $O(\log\log n)$ and $O(n)$ operations.

We comment on how to actually perform step (c) using $n/\log\log n$ processors within $O(\log\log n)$ time. Step [illegible] will employ the $O(\log\log l)$ time optimal speed up algorithm for finding the minimum among l elements given in an array of length l, as in Shiloach & Vishkin [SV81]. The processors allocation for Step (c) is done as follows. In order to simplify the explanation, assume that we have $3n/\log\log n$ processors. We assign three processors to each successive subarray (called *interval*) of length $\log\log n$ in array $LABEL$. These three processors will participate in the minimum computation of all subarrays that intersect their interval. If there is a subarray that ends in the interval, we assign to it the first processor. If there are subarrays that are contained in the interval, we assign to all of them the second processor. If there is a subarray that begins in the interval, we assign to it the third processor. If the interval is fully contained in a subarray assign to it the first processor (and the other two will remain idle). Each processor of each interval clearly finishes finding the (local) minima for the intersection of the interval with each intersecting subarray in $O(\log\log n)$ time. The first and third processors may then participate in a global minimum computation of an appropriate subarray. Such global computation will apply the Shiloach-Vishkin algorithm for each subarray separately.

The space complexity is dominated by the array BB whose size is m. The values written in the memory locations $BB[t_i]$ in Step (a) are from the range $[1..n]$. Therefore, the sorting problem of step (b) is indeed of elements from this range only. The Lemma follows. ∎

A major drawback of the above reduction is the potentially large space that it might require. (Recall that this space is in addition to the memory which is as in the simulated min-CRCW.) However, the space consuming array BB is only used in step (a). It is easy to see that the parallel hashing scheme from section 3 can be applied here. As a result the additional space which is used by the reduction is reduced to $O(n)$ while the time complexity remains the same (up to a constant factor), only expected rather than deterministic, using the same number of processors.

Thus, Simulation Result 1 follows from Lemma 15, Lemma 9 and Theorem 2.

Comments:

1. Simulation result 1 improves a similar theorem in the survey of Eppstein and Galil [EG88] (the min-CRCW is called there strong-CRCW), where it is assumed that addresses can be written in at most $O(\log n)$ bits. Recall that Simulation result 1 does not impose any size restriction on the memory to be used by the simulated machine.

2. Lemma 15 can be extended to hold in $O(\log n / \log\log n)$ times for a fetch&add step of a fetch&add-CRCW PRAM. This extension, Lemma 9 and Theorem 2 imply the following extensions of simulation result 1: A single fetch&add step of n processors on a fetch&add-CRCW PRAM can be simulated by $\frac{n}{\log n}$ arbitrary-CRCW PRAM processors in $O(\log n)$ expected time and $O(n)$ space.

Simulation Result 1 states that a single step of an n-processor min-CRCW can be simulated with $\Theta(n)$ expected number of operations by an arbitrary-CRCW PRAM. To see what can be done deterministically we first state the following lemma due to Hagerup:

Lemma 16 *([Hag87]) For any fixed $\epsilon > 0$, n integers of size polynomial in n can be sorted in $O(\log n)$ time by a priority-CRCW PRAM using $O(\frac{n \log\log n}{\log n})$ processors and $O(n^{1+\epsilon})$ space.*

Following Lemma 15 and Lemma 16 we have

Simulation Result 2 A single step of n processors on a min-CRCW PRAM with memory size S can be simulated by $\frac{n \log\log n}{\log n}$ priority-CRCW PRAM processors in $O(\log n)$ time and $O(S + n^{1+\epsilon})$ space (for any fixed $\epsilon > 0$).

Comment 3. Simulation Result 2 can be extended for a fetch&add-CRCW PRAM similarly to Comment 2 above.

Two more simulation results are stated here. They are based on [CDHR88], as explained in Appendix B.

Simulation Result 3 An n-processor min-CRCW that uses memory of size S, where each memory cell contains a $\log m$-bit word, can be simulated by an n-processor arbitrary-CRCW PRAM with a slowdown of $O(\log\log m)$, using $O(mS)$ space.

Simulation Result 4 An n-processor min-CRCW that uses memory of size S, where each memory cell contains a $\log m$-bit word, can be simulated by an $n \log m$-processor common-CRCW PRAM in $O(1)$ time, using $O(mS)$ space.

4.1 Applications

In this section we apply some of the simulations to the parallel DNN and Sorting algorithms and derive complexity results for standard CRCW PRAM models.

Recall that in algorithm DNN there are $O(\log\log m)$ phases. Each phase takes $O(1)$ time for n min-CRCW processors. Following Simulation Results 1, 2, 3 and 4 we have

Corollary 17 *Algorithm DNN and its extension can be implemented as follows:*

- *On arbitrary-CRCW in $O(\log n \log\log m)$ expected time, $O(n \log\log m)$ expected number of operations and $O(n)$ space.*
- *On priority-CRCW in $O(\log n \log\log m)$ time, $O(n \log\log m \log\log n)$ operations and $O(m^\epsilon)$ space, for any fixed $\epsilon > 0$.*
- *On arbitrary-CRCW in $O((\log\log m)^2)$ time, $O(n(\log\log m)^2)$ operations and $O(m^\epsilon)$ space, for any fixed $\epsilon > 0$.*
- *On common-CRCW in $O(\log\log m)$ time, $O(n \log m \log\log m)$ operations and $O(m^\epsilon)$ space, for any fixed $\epsilon > 0$.*

Proof: The first implementation is as follows. Recall that the opening sentence of Section 2.4 explains how to trivially extend the DNN algorithm for non-distinct elements using $O(nm)$ variables. At each phase of algorithm DNN, Theorem 2 is used to map the $O(nm)$ variables into $O(n)$ space in $O(\log n)$ expected time and $O(n)$ expected number of operations. The step of the min-CRCW PRAM is then done, using Simulation Result 1, in $O(\log n)$ expected time and $O(n)$ expected number of operations, using $O(n)$ space. There are $O(\log\log m)$ phases by Lemma 4.

For the other implementations we use the reduced space DNN algorithm of Lemma 6. Thus, at each phase we implement a step of min-CRCW PRAM with $O(m^{\epsilon/2})$ space, for any fixed $\epsilon > 0$. We use Simulations Results 2,3 and 4 to implement the second, third and forth implementations, respectively. Note that the values written by the min-CRCW PRAM are at most $O(m^{\epsilon/2})$. ∎

Recall that after running algorithm DNN, the Sorting algorithm can be finished by using the List Ranking procedure. The latter takes $O(\log n)$ time, $O(n)$ operations and $O(n)$ space on EREW PRAM. Thus, Sorting can be implemented with the same complexities as stated in Corollary 17 except for an increase in time to $O(\log n)$ in the last 2 cases.

In particular, the first item in Corollary 17 implies the following parallel sorting result.

Theorem 3 *Sorting n integers from the range $[1..m]$ can be done on a randomized arbitrary-CRCW in $O(\log n \log\log m)$ expected time, $O(n \log\log m)$ expected number of operations and $O(n)$ space.*

5 Conclusion

We gave an $o(n \log n)$ time randomized algorithm for sorting integers drawn from a super-polynomial range. Our algorithm takes $O(n \log\log m)$ expected time and $O(n)$ space. A parallel version of the algorithm achieves optimal speed up.

Sorting is a fundamental problem in computer science, therefore we expect these results to have applications in quite a few problems. We outline one possible direction for such applications. Consider a problem that is defined on an m by m grid of points, and suppose an algorithm for the problem needs sorting of points along the x or y axes. Our integer sorting algorithm is likely to be helpful.

An open question is whether a space efficient deterministic Integer Sorting algorithm in $o(n \log n)$ time can be found, for integers drawn from a superpolynomial range $[1..n^{polylog(n)}]$.

The second main topic is the parallel hashing technique. It achieves optimal speed up and takes expected logarithmic time. The parallel hashing technique enables drastic reduction of space requirements for the price of using randomness. The technique was used in the parallel sorting algorithm and in the simulation results; its applicability to other problems was demonstrated.

An open question: design an optimal parallel speed up hashing scheme $F : W \rightarrow [1..O(n)]$ that takes sublogarithmic time.

A third topic of independent interest is the efficient simulation of strong models of CRCW PRAM by more acceptable ones, and the methodology of designing parallel algorithms for the strong models with automatic simulations on acceptable models. We expect to see more applications of this approach in the design of parallel algorithms.

Acknowledgment: We wish to thank Yehuda Afek, Noga Alon, Eli Gafni, Zvi Galil and Hanoch Levy for stimulating discussions.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley Publishing Company, 1974.

[AIL$^+$88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree. *Algorithmica*, 3:347–365, 1988.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(nlogn)$ sorting network. In *Proc. of the 15th Ann. ACM Symp. on Theory of Computing*, pages 1–9, 1983.

[AM88] R.J. Anderson and G.L. Miller. Optimal parallel algorithms for list ranking. In *3rd Aegean workshop on computing, Lecture Notes in Computer Science 319, 1988, Springer-Verlag*, pages 81–90, 1988.

[Apo84] A. Apostolico. The myriad virtues of subword trees. *in "Combinatorial Algorithms on Words" (A. Apostolico and Z. Galil, Eds.), NATO ASI Series F, Vol. 12,*, pages 85–96, 1984.

[BDH+89] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. Technical Report TR 15/1989, Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, W. Germany., November 1989.

[Bre74] R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.*, 21:302–206, 1974.

[CDHR88] B.S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write PRAM models. In *Mathematical Foundations of Computer Science '88*, 1988.

[Col86] R. Cole. Parallel merge sort. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 511–516, 1986.

[CV86] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 478–491, 1986.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17:128–142, 1988.

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Control*, 81:334–352, 1989.

[DadH89] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *Proc. of the 1989 Symposium on Parallel Algorithms and Architectures*, pages 360–368, 1989.

[EG88] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233–283, 1988.

[FKS84] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. of the Association for Computing Machinery*, 31:538–544, 1984.

[FRW88] F.E. Fich, P.L. Ragde, and A. Wigderson. Simulations among concurrent-write PRAMs. *Algorithmica*, 3:43–51, 1988.

[Gal84] Z. Galil. Optimal parallel algorithms for string matching. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 240–248, 1984.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *J. of Complexity*, 4:33–72, 1988.

[Hag87] T. Hagerup. Towards optimal parallel bucket sorting. *Information and computation*, 75:39–51, 1987.

[Joh82] D.B. Johnson. A priority queue in which initialization and queue operations take $O(\log\log D)$ time. *Math. Systems Theory*, 15:295–309, 1982.

[Knu73] D.E. Knuth. *The art of computer programming*, volume 3, Sorting and searching. Addison-Wesley, Reading, 1973.

[KR84] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

[KR88] R.M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division (EECS) U. C. Berkeley, 1988.

[KRS88] C.P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. In *Proc. 15th ICALP, Springer Verlag Lecture Notes in Computer Science 317*, pages 333–346, 1988.

[KU86] A. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. In *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, pages 160–168, 1986.

[LW88] Y. Lamdan and H.J. Wolfson. Geometric hashing: a general and efficient model-based recognition scheme. In *Proc. 2nd Intl' Conf. on Computer Vision, Tampa, FL*, pages 238–249, 1988.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[PS80] W.J. Paul and J. Simon. Decision trees and random access machines. In *Symp. uber Logic und Algorithmik*, 1980.

[Ran87] A.G. Ranade. How to emulate shared memory. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 185–194, 1987.

[RR89] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18:594–607, 1989.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms*, 2:88–102, 1981.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Vis83] U. Vishkin. Synchronous parallel computation - a survey. Technical Report TR 71, Dept. of Computer Science, Courant Institute, New York University, 1983.

A Proof of Proposition 1

This appendix gives a proof of Proposition 1 for completeness of the presentation. We suggest not to include this appendix in a published version of the paper.

Proposition 1 Let $x[j]$ be the left neighbor of $x[i]$. Then for each level of the recursion the following properties hold:

(a) The values of all elements in L are distinct and the values of all elements in R are distinct.

(b) $x_l[i]$ and $x_r[j]$ are both represented in the same recursive sub-problem.

(c) $x_r[j] < x_l[i]$.

(d) $x_r[j]$ is the left neighbor of $x_l[i]$.

(e) Each element is represented in exactly one recursive sub-problem.

(f) Set L is nonempty if and only if set R is nonempty.

Proof:

The proof is by induction on the level of the recursion.

Inductive Base: Trivial.

Induction step:

(a): After step 2 the values of elements in L_k and in R_k are unchanged and therefore, by the inductive hypothesis on (a), we only need to look at the sets GL and GR. In step 2, for each k at most one element from L_k is selected into GL, and at most one element from R_k is selected into GR. These (possibly) selected elements are the only ones that are assigned with the value k.

(b): Following from the inductive hypothesis on (b), before step 1 elements $x_l[i]$ and $x_r[j]$ are both represented in the same recursive sub-problem. Assume that after step 1 $x_l[i] \in L_k$ and $x_r[j] \in R_{k'}$ for some k and k'. If $x_l[i]$ was selected into GL, i.e. $x_l[i]$ is the smallest element in $L_k \bigcup R_k$, then $x_r[j]$ was selected into GR, i.e. $x_r[j]$ is the largest element in $L_{k'} \bigcup R_{k'}$. Otherwise, we have from the inductive hypothesis on (a) and (c) that $x_r[j] < b_{k'} < x_l[i]$ (before $b_{k'}$ was changed in set GR) which contradicts the inductive hypothesis on (d). Similarly, if $x_r[j]$ was selected into GR then $x_l[i]$ was selected into GL, otherwise we have from the inductive hypothesis on (a) and (c) that $x_r[j] < a_k < x_l[i]$, which contradicts the inductive hypothesis on (d). If both $x_l[i]$ and $x_r[j]$ were not selected into GL and GR, respectively, then $k = k'$ otherwise by the inductive hypothesis on (a) and (c) we have $x_r[j] < a_k < x_l[i]$ which contradicts the inductive hypothesis on (d). Thus, $x_l[i]$ and $x_r[j]$ are in GL and GR, respectively, or in L_k and R_k, respectively.

(c): Following from the inductive hypothesis on (b), if $x_l[i]$ and $x_r[j]$ are in L_k and R_k, respectively, then their values are unchanged. If they are in GL and in GR, respectively, then after step 2 $x_l[i] = k$ and $x_r[j] = k'$ (where after step 1 $x_l[i] \in L_k$ and $x_r[j] \in R_{k'}$). Following from the definition of L_k and $R_{k'}$ in step 1 and from the inductive hypothesis on (c) we have $k < k'$.

(d): The values of all elements in sets L_k and R_k are unchanged. Thus, following from the inductive hypothesis on (b) and (d), if $x_l[i] \in L_k$ and $x_r[j] \in R_k$ then $x_r[j]$ is the left neighbor of $x_r[j]$. We only need to deal with the case that $x_l[i] \in GL$ and $x_r[j] \in GR$. Assume, by negation, that there is an element $y \in GR$ such that $x_r[j] < y < x_l[i]$. Clearly, after step 1 we had $x_l[i] \in L_k$, $x_r[j] \in R_{k'}$ and $y \in R_{k''}$ with $k' < k'' < k$, contradicting the inductive hypothesis on (d).

(e): Following from the inductive hypothesis on (e), in step (2) each element $x_l[i]$ is either in L_k for some k or in GL. Similarly, each element $x_r[j]$ is either in $R_{k'}$ for some k' or in GR. Thus, in step (3) each element is represented in exactly one recursive sub-problem.

(f): Following from the inductive hypothesis on (b). ∎

B Simulation Results 3 and 4

This appendix gives a proof of Simulation Results 3 and 4 for completeness of the presentation. We suggest not to include this appendix in a published version of the paper.

Consider some CRCW machine. We say that a memory cell *contains* a processor if this processor has the address of this cell written in a special local register. Each processor is contained in at most one cell. A cell that contains at least one processor is said to be *non-empty*, otherwise it is said to be *empty*. The *Find-First* problem of size n is defined as follows: Given is an array $A[1..n]$, each cell of which is empty or contains exactly one processor, find the lowest-numbered non-empty cell of A. We similarly define the *Extended Find-First* problem of size n: Given is an array $A[1..n]$, each cell of which is empty or contains one or more processors, find the lowest-numbered non-empty cell of A.

Chlebus, Diks, Hagerup and Radzik [CDHR88] show how to simulate priority-CRCW PRAM on weaker CRCW PRAMs. Their simulations are based on solving the *Find-First* problem[2]. Specifically, a 'write' step of n-processor priority-CRCW is done as follows. For each memory cell c to which there is at least one processor that attempts to write, a Find-First problem of size n is defined: $A_c[i]$ contains processor P_i if P_i attempts to write into cell c. The space required for the simulating machine is $O(Sn)$ where S is the size of memory being used by the simulated priority-CRCW PRAM. Time requirements are discussed later.

Let *min-CRCW*(S, m) be a min-CRCW PRAM with space S and m being an upper bound on the values that can be written into the memory cells. A 'write' step of min-CRCW(S, m) PRAM can be done, similarly to the above, by using S Extended Find-First problems of size m. For each memory cell c to which there is at least one processor that attempts to write, an Extended Find-First problem of size m is defined: $A_c[i]$ contains processor P_j if P_j attempts to write i into cell c. The space required for the simulating machine is $O(Sm)$. A 'write' step of the min-CRCW(S, m) PRAM can be also reduced to S Find-First problems of size nm. Specifically, $A_c[< i, j >]$ contains processor P_j if P_j attempts to write i into cell c (here 'lowest number' is with respect to the lexicographic ordering). The space required for the simulating machine here is $O(Snm)$.

Chlebus et al. show how to solve the Find-First problem of size m on several machines:

Proposition 18 *([CDHR88]) The Find-First problem of size m can be solved as follows: (a) On arbitrary-CRCW PRAM in $O(\log\log m)$ time; (b) On common-CRCW PRAM in constant time, provided that each processor has additional $\log m$ processors.*

The algorithms in [CDHR88] solve also the Extended Find-First problem since all processors contained in the same cell act identically (independently of their index). This is enough when using the arbitrary-CRCW and the common-CRCW PRAMs. Using the above we have

Simulation Result 3 An n-processor min-CRCW(S, m) PRAM can be simulated by an n-processor arbitrary-CRCW PRAM with $O(\log\log m)$ slowdown, using $O(mS)$ space.

Simulation Result 4 An n-processor min-CRCW(S, m) PRAM can be simulated by an $n\log m$-processor common-CRCW PRAM in $O(1)$ time, using $O(mS)$ space.

[2]Fich, Ragde and Wigderson [FRW88] also used the Find-First problem (there called "leftmost prisoner problem") to simulate priority-CRCW on common-CRCW PRAM.